L'ESTIMATION DE DISTRIBUTION À L'AIDE D'UN AUTOENCODEUR

par

Mathieu Germain

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 8 juin 2015

Sommaire

Ce mémoire introduit MADE, un nouveau modèle génératif spécifiquement développé pour l'estimation de distribution de probabilité pour données binaires. Ce modèle se base sur le simple autoencodeur et le modifie de telle sorte que sa sortie puisse être considérée comme des probabilités conditionnelles. Il a été testé sur une multitude d'ensembles de données et atteint des performances comparables à l'état de l'art, tout en étant plus rapide. Pour faciliter la description de ce modèle, plusieurs concepts de base de l'apprentissage automatique seront décrits ainsi que d'autres modèles d'estimation de distribution.

Comme son nom l'indique, l'estimation de distribution est simplement la tâche d'estimer une distribution statistique à l'aide d'exemples tirés de cette dernière. Bien que certains considèrent ce problème comme étant le Saint Graal de l'apprentissage automatique, il a longtemps été négligé par le domaine puisqu'il était considéré trop difficile. Une raison pour laquelle cette tâche est tenue en si haute estime est qu'une fois la distribution des données connue, elle peut être utilisée pour réaliser la plupart des autres tâches de l'apprentissage automatique, de la classification en passant par la régression jusqu'à la génération.

L'information est divisée en trois chapitres principaux. Le premier donne un survol des connaissances requises pour comprendre le nouveau modèle. Le deuxième présente les précurseurs qui ont tenu le titre de l'état de l'art et finalement le troisième explique en détail le modèle proposé.

Mots-clés: réseau de neurones ; autoencodeur ; apprentissage automatique ; apprentissage non-supervisé ; architecture profonde, estimation de distribution

Remerciements

Je débute ce mémoire en exprimant ma très grande gratitude à tous les gens qui ont contribué à rendre ma vie excitante et agréable au cours de ces quelques années. Merci aussi à tous mes collègues de l'Université de Sherbrooke pour le temps passé avec eux et à toutes les personnes venues en échange de pays éloignés pour enrichir la vie à Sherbrooke.

Il est tout à fait naturel que je remercie particulièrement ceux qui ont participé à l'accomplissement de ma maîtrise. Premièrement, mon superviseur Hugo Larochelle, pour m'avoir accueilli dans son laboratoire et m'avoir donné une multitude d'opportunités académiques. Merci à Max, Mik, Luc et Adam pour les discussions mathématiques interminables tirant sur la philosophie. Merci à JF pour les rapides diagnostics de la grappe de calculs et le support technique. Un grand merci à Marco pour avoir toujours été prêt à échanger sur des idées, ce qui m'a grandement aidé à avancer. Finalement, merci à Nil, Vanessa, Gab, Hugo, JF, Marco et Marie pour m'avoir aidé à rendre ce mémoire plus facile à lire et le plus informatif possible.

Pour terminer, un petit merci spécial à ma famille qui me soutient depuis toujours et à tous ceux que je n'ai pas eu la chance de remercier ici.

-Mathieu

Abréviations

Nom de modèles :

MADE *Masked Autoencoder for Distribution Estimation*

RBM *Restricted Boltzmann Machine*

NADE *Neural Autoregressive Distribution Estimator*

DARN *Deep AutoRegressive Networks*

Table des matières

Sommaire **i**

Remerciements **ii**

Abréviations **iii**

Table des matières **iv**

Liste des figures **vi**

Liste des algorithmes **viii**

1 Introduction **1**

1.1 Apprentissage automatique 1

1.1.1 Apprentissage supervisé 2

1.1.2 Apprentissage non-supervisé 2

1.2 Réseau de neurones artificiels 2

1.2.1 Perceptron Multicouche 3

1.2.2 Autoencodeur 6

1.3 Entraînement 6

1.3.1 Descente de gradient 6

2 Précurseurs **10**

2.1 RBM 10

2.2 NADE 12

2.3 Deep NADE & Orderless NADE 13
2.4 DARN 15

3 Masked Autoencoder for Distribution Estimation 16
3.1 Abstract 16
3.2 Introduction 17
3.3 Autoencoders 18
3.4 Distribution Estimation as Autoregression 19
3.5 Masked Autoencoders 20
3.5.1 Deep MADE 22
3.5.2 Order-agnostic training 24
3.5.3 Connectivity-agnostic training 25
3.6 Related Work 26
3.7 Experiments 28
3.7.1 UCI evaluation suite 29
3.7.2 Binarized MNIST evaluation 30
3.8 Conclusion 34

Conclusion 36

Liste des figures

1.1 Illustration haut niveau d'un perceptron multicouche. $\boldsymbol{x}$ représente les entrées, $\boldsymbol{h^l}$ la l-ième couche cachée et $\boldsymbol{y}$ les sorties, le tout interconnecté par les poids $\boldsymbol{W^l}$. 3
1.2 Perceptron multicouche. x_i représente les différentes entrées du réseau, h_i^l les neurones de la l-ième couche cachée et y_i les sorties, le tout interconnecté par les matrices de poids $\boldsymbol{W^l}$. 4
1.3 Démontre le fonctionnement interne d'un neurone. h_i^l représente la valeur en sortie de ces neurones, W_i^l est la force de la connexion avec l'$i$ème neurone de la couche précédente $l-1$. Toutes les multiplications entre les forces de connexion W_i^l et les valeurs de sortie de la couche précédente sont additionnées entre elles pour être ensuite modifiées par la fonction d'activation σ. 5
1.4 Example de fonctions d'activation. 5

2.1 *Restricted Boltzmann Machine* avec 3 unités visibles x_i et 3 unités cachées h_i. 11
2.2 NADE avec 4 entrées et 3 neurones cachés (répété pour chaque entrée). Les groupes de poids de même couleur sont dits liés, ce qui implique que les mêmes poids sont réutilisés (les liens en noir ne sont pas liés). Les biais sont omis du diagramme par soucis de clarté. 13
2.3 Deep NADE avec 4 entrées, 3 neurones par couche cachée et 2 couches cachées. Les groupes de poids de même couleur sont dits liés (les liens en noir ne sont pas liés). Les biais sont omis du diagramme par soucis de clarté. 14
2.4 Exemples d'entraînements partiels pour Orderless NADE. 15

3.1 **Left : Conventional three hidden layer autoencoder**. Input in the bottom is passed through fully connected layers and point-wise nonlinearities. In the final top layer, a reconstruction specified as a probability distribution over inputs is produced. As this distribution depends on the input itself, a standard autoencoder cannot predict or sample new data. **Right : MADE**. The network has the same structure as the autoencoder, but a set of connections is removed such that each input unit is only predicted from the previous ones, using multiplicative binary masks ($\mathbf{M}^{\mathbf{W}^1}$, $\mathbf{M}^{\mathbf{W}^2}$, $\mathbf{M}^{\mathbf{V}}$). In this example, the ordering of the input is changed from 1,2,3 to 3,1,2. This change is explained in section 3.5.2, but is not necessary for understanding the basic principle. The numbers in the hidden units indicate the maximum number of inputs on which the kth unit of layer l depends. The masks are constructed based on these numbers (see Equations 3.12 and 3.13). These masks ensure that MADE satisfies the autoregressive property, allowing it to form a probabilistic model, in this example $p(\mathbf{x}) = p(x_2)\,p(x_3|x_2)\,p(x_1|x_2, x_3)$. Connections in light gray correspond to paths that depend only on 1 input, while the dark gray connections depend on 2 inputs. 23
3.2 Impact of the number of masks used with a single hidden layer, 500 hidden units network, on binarized MNIST. 32
3.3 Left : Samples from a 2 hidden layer MADE. Right : Nearest neighbour in binarized MNIST. 34

Liste des algorithmes

1 Computation of $p(\mathbf{x})$ and learning gradients for MADE with order and connectivity sampling. D is the size of the input, L the number of hidden layers and K the number of hidden units. 35

Chapitre 1

Introduction

Ce chapitre contient les notions essentielles à la bonne compréhension de ce mémoire. Les différentes sections introduisent plusieurs concepts : l'apprentissage automatique (Section 1.1), les réseaux de neurones artificiels (Section 1.2) et leur entraînement (Section 1.3).

1.1 Apprentissage automatique

L'apprentissage automatique est un sous-domaine de l'informatique issu de l'intelligence artificielle. Cette discipline scientifique est principalement divisée en deux volets, soit l'apprentissage supervisé et non-supervisé. Tout en étant très fortement liée aux statistiques et à l'optimisation mathématique, elle explore la construction de modèles et l'étude d'algorithmes pouvant apprendre à partir de données. Ces données sont spécifiques à une tâche précise que le modèle essaie de résoudre en apprenant à partir d'exemples. Le spectre de tâches que l'apprentissage automatique aide à résoudre est extrêmement large : la vision par ordinateur, le traitement automatique de la langue, les engins de recherche, les diagnostics médicaux, la détection de fraude, le contrôle de robots, l'analyse des marchés financiers, etc.

1.1.1 Apprentissage supervisé

Dans le domaine de l'apprentissage automatique, l'apprentissage supervisé est la tâche d'inférer une fonction à partir de données étiquetées. Ces données d'entraînement constituent un ensemble d'exemples composés d'un objet d'entrée, généralement un vecteur, et d'une valeur de sortie désirée, son étiquette. Un algorithme d'apprentissage supervisé analyse les données d'entraînement et produit une fonction, laquelle peut être utilisée pour générer une sortie sur de nouveaux exemples. Un scénario optimal permettra à l'algorithme de bien généraliser ce qui a été appris lors de l'entraînement, c'est-à-dire, de déterminer correctement les étiquettes pour des instances qui n'ont jamais été observées auparavant.

La tâche la plus commune en apprentissage supervisé est la classification. Un exemple classique est la classification de l'ensemble de données MNIST [Yann LeCun, 1998] qui se compose d'images représentant des chiffres manuscrits. La tâche de classification à effectuer dans ce cas est d'assigner une classe – le chiffre que l'image représente – à chacun des éléments de l'ensemble.

1.1.2 Apprentissage non-supervisé

L'apprentissage non-supervisé, quant à lui, tente de trouver une structure cachée dans des données non étiquetées. Ce type d'apprentissage englobe de nombreuses techniques visant soit à résumer, à expliquer les principales caractéristiques ou à déterminer la distribution d'un ensemble de données.

Différentes méthodes incluses dans l'apprentissage non-supervisé comprennent le partitionnement de données, les modèles de Markov cachés, la réduction de dimensionnalité, l'estimation de distribution et bien d'autres.

1.2 Réseau de neurones artificiels

Les réseaux de neurones artificiels sont une famille d'algorithmes d'apprentissage statistique légèrement inspirés par les réseaux de neurones biologiques et sont utilisés pour estimer ou

approximer des fonctions, généralement inconnues, pouvant dépendre d'un grand nombre d'entrées. Les réseaux de neurones artificiels sont généralement présentés comme des systèmes de «neurones» interconnectés pouvant calculer des valeurs à partir des différentes entrées de ceux-ci.

1.2.1 Perceptron Multicouche

Un perceptron multicouche [Bishop, 2006, Chapter 5] est un réseau de neurones artificiels qui génère une sortie à partir d'une entrée. Il se compose de plusieurs neurones organisés en couches dans un graphe orienté, chaque couche entièrement connectée à la suivante. Ces neurones, à l'exception des neurones d'entrée, se composent d'une transformation linéaire suivi d'une fonction d'activation non linéaire (Figure 1.3). Ce modèle est une modification du perceptron linéaire standard [Rosenblatt, 1957] et peut distinguer des données qui ne sont pas linéairement séparables.

Les figures 1.1, 1.2, 1.3 illustrent, avec différents niveaux de détails, le perceptron multicouche. Notez que dans ce mémoire, les matrices sont représentées par des lettres majuscules en gras et les vecteurs, par des lettres minuscules en gras.

La figure 1.1 illustre, à très haut niveau, le perceptron multicouche. Les entrées du réseau sont représentées par $\boldsymbol{x}$, les $\boldsymbol{h^l}$ représentent la l-ième couche cachée composée de neurones et $\boldsymbol{y}$ les sorties, le tout interconnecté par les matrices de poids $\boldsymbol{W^l}$.

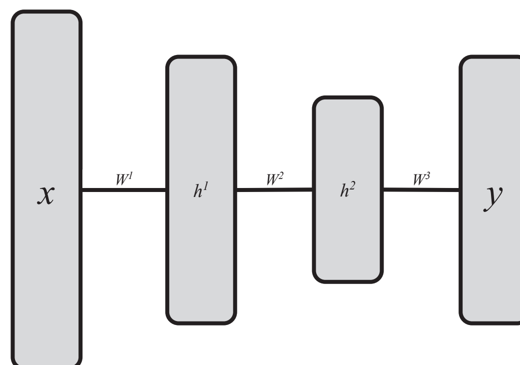


Figure 1.1 – Illustration haut niveau d'un perceptron multicouche. $\boldsymbol{x}$ représente les entrées, $\boldsymbol{h^l}$ la l-ième couche cachée et $\boldsymbol{y}$ les sorties, le tout interconnecté par les poids $\boldsymbol{W^l}$.

Plus en détails, la figure 1.2 expose comment les couches cachées et les poids sont structurés. x_i représente les différentes entrées du réseau, h_i^l les neurones de la l-ième couche cachée et y_i les sorties, encore une fois interconnectées par les matrices de poids $\boldsymbol{W^l}$.

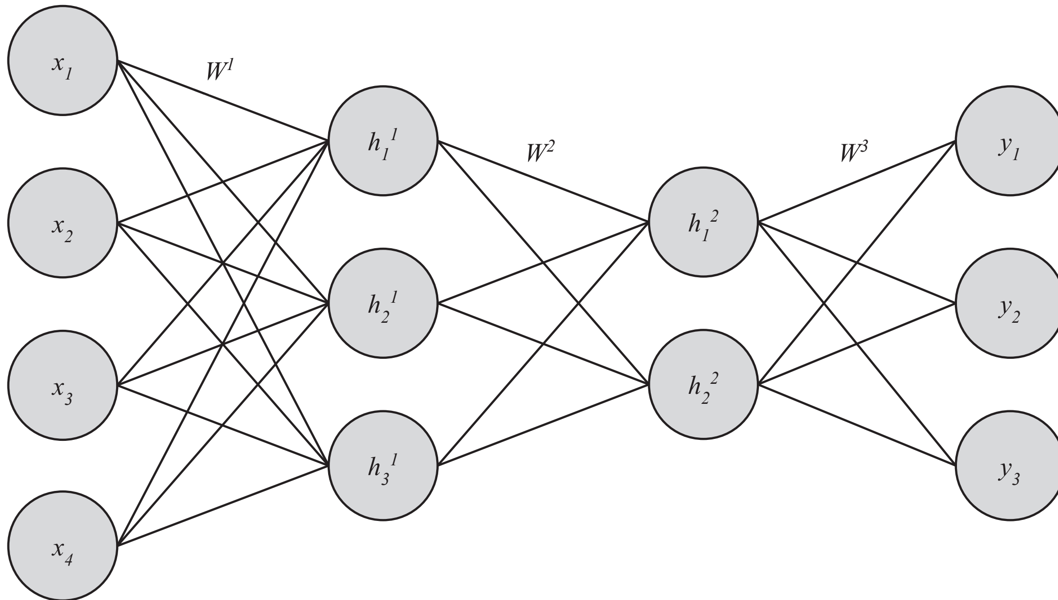


Figure 1.2 – Perceptron multicouche. x_i représente les différentes entrées du réseau, h_i^l les neurones de la l-ième couche cachée et y_i les sorties, le tout interconnecté par les matrices de poids $\boldsymbol{W^l}$.

Dans la figure 1.3, le fonctionnement interne d'un neurone est expliqué en détail. h_i^l représente la valeur en sortie de ces neurones, W_i^l est la force de la connexion avec l'$i$ème neurone de la couche précédente $l-1$. Toutes les multiplications entre les forces de connexion W_i^l et les valeurs de sortie de la couche précédente sont additionnées entre elles pour être ensuite modifiées par la fonction d'activation σ. Cette fonction d'activation peut varier en fonction des besoins, mais les plus utilisées sont la sigmoïde (Figure 1.4a) et la tangente hyperbolique communément appelée tanh (Figure 1.4b).

Par souci d'efficacité, les opérations dans le neurone de la figure 1.3 sont effectuées de façon vectorielle, par couche. Chacune des couches est calculée comme suit : $\boldsymbol{h^l} = \sigma(\boldsymbol{W^l}\,\boldsymbol{h_{l-1}})$. Dans le cas du calcul de la première couche, le $\boldsymbol{h^0}$ est le $\boldsymbol{x}$ d'entrée.

Maintenant que le fonctionnement de base est bien établi, un détail peut être ajouté pour

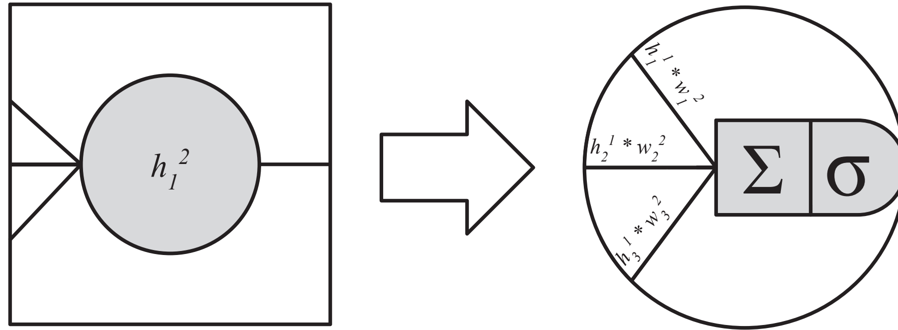
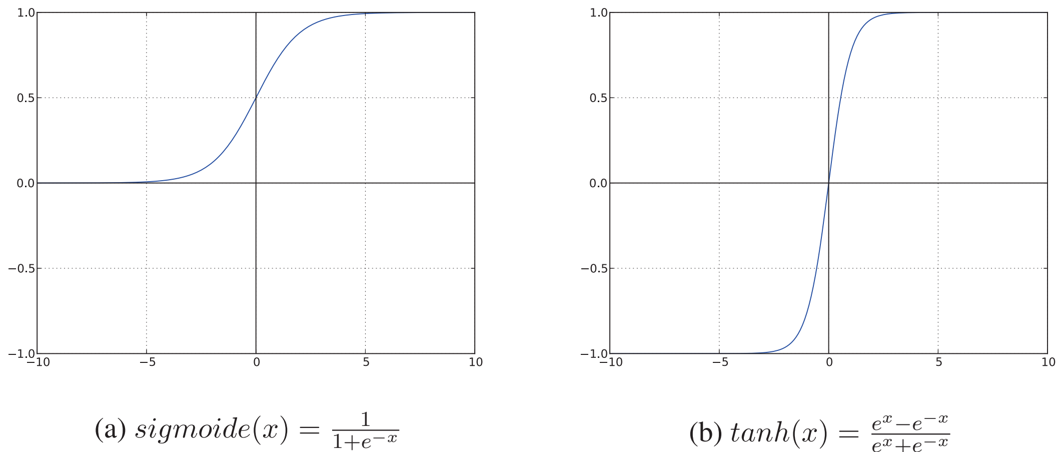


Figure 1.3 – Démontre le fonctionnement interne d'un neurone. h_i^l représente la valeur en sortie de ces neurones, W_i^l est la force de la connexion avec l'$i$ème neurone de la couche précédente $l-1$. Toutes les multiplications entre les forces de connexion W_i^l et les valeurs de sortie de la couche précédente sont additionnées entre elles pour être ensuite modifiées par la fonction d'activation σ.



(a) $sigmoide(x) = \frac{1}{1+e^{-x}}$

(b) $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Figure 1.4 – Example de fonctions d'activation.

donner plus de flexibilité à la fonction d'activation. Ce détail est un terme de biais b_i^l que chacun des neurones possède, qui transforme l'équation précédente en $\boldsymbol{h^l} = \sigma(\boldsymbol{W^l}\ \boldsymbol{h^{l-1}} + \boldsymbol{b^l})$. Ce terme est nécessaire pour tenir compte de la possibilité d'un décalage de la moyenne. Sans le biais la solution apprise doit passer par l'origine.

La formule complète pour calculer la sortie du réseau en 1.2, communément appelée *la*

propagation avant, serait :

$$\boldsymbol{y} = \sigma(\boldsymbol{W^3}\,\sigma(\boldsymbol{W^2}\,\sigma(\boldsymbol{W^1}\boldsymbol{x} + \boldsymbol{b^1}) + \boldsymbol{b^2}) + \boldsymbol{b^3}) \tag{1.1}$$

1.2.2 Autoencodeur

L'autoencodeur [P. Baldi, 1989] peut être vu comme un cas particulier du perceptron multicouche où la valeur en entrée et la valeur désirée en sortie sont la même. Ceci implique qu'il y a autant de neurones en entrée qu'en sortie. En le configurant ainsi, le réseau de neurones doit donc apprendre à reconstruire ce qu'il reçoit en entrée. Dans le cas où les $\boldsymbol{y}$ sont une reconstruction des $\boldsymbol{x}$, ils sont dénotés $\widehat{\boldsymbol{x}}$ pour souligner ce fait.

1.3 Entraînement

Considérant que les paramètres θ (les $\boldsymbol{W}$ et $\boldsymbol{b}$ dans l'équation 1.1) sont initialisés aléatoirement, le $\boldsymbol{y}$ calculé sera de piètre qualité. Pour y remédier, le réseau doit donc être entraîné. Ceci consiste à modifier les paramètres de ce réseau en lui présentant des exemples d'entraînement jusqu'à ce que les prédictions $\boldsymbol{y}$ soient d'une qualité satisfaisante. Cette mesure de qualité, qui permet de quantifier les erreurs du réseau, est souvent appelée la *fonction de coût* ou *fonction objectif*. Un exemple simple de fonction objectif est l'erreur quadratique moyenne (Équation 1.2). Dans cette fonction, N représente le nombre total d'exemples d'entraînement, $\boldsymbol{t_n}$ représente la valeur attendue pour un exemple donné $\boldsymbol{x_n}$ et $\boldsymbol{y_n}$ est la valeur de sortie du réseau pour un exemple donné.

$$\ell = \frac{1}{N}\sum_{n=1}^{N} \|\boldsymbol{t_n} - \boldsymbol{y_n}\|^2 \tag{1.2}$$

1.3.1 Descente de gradient

L'algorithme le plus commun pour entraîner des réseaux de neurones artificiels est la descente de gradient. Cet algorithme minimise la fonction objectif par rapport aux paramètres

du réseau.

La descente de gradient procède en répétant une mise à jour des paramètres, visant à minimiser la fonction objectif. Une mise à jour se décompose en 3 étapes :

1. Calculer la prédiction du réseau sur les exemples d'entraînement (Équation 1.1).
2. Déterminer les modifications à effectuer au réseau par rapport à l'erreur calculée par la fonction objectif. Plus précisément, calculer le gradient de la fonction objectif par rapport aux paramètres du réseau à l'aide de l'algorithme de *rétro-propagation du gradient* [Bishop, 2006, Chapter 5.3].
3. Mettre à jour les paramètres du réseau d'un facteur α, appelé taux d'apprentissage, du gradient calculé en 2 (Équation 1.3).

Ces étapes sont répétées jusqu'à ce que la condition d'arrêt soit atteinte. La décision d'arrêter l'apprentissage peut être prise de différentes façons. Par exemple, après un nombre prédéfini d'itérations ou bien lorsque le réseau ne s'améliore plus suffisament.

Il est aussi important de noter que, par abus de langage, certains utilisent le terme *rétro-propagation du gradient* incorrectement pour référer à la fois à l'algorithme de descente du gradient et à la *rétro-propagation du gradient*.

Il existe trois variantes principales affectant l'étape 1 de la descente de gradient. La première, décrite dans la précédente énumération, communément appelée descente de gradient *batch*, calcule une prédiction sur tous les exemples d'entraînement. La plus populaire, la descente de gradient *mini-batch*, consiste à utiliser seulement un sous-ensemble aléatoire des exemples d'entraînement chaque fois qu'une mise à jour est effectuée. La popularité de cette variante est en grande partie liée au fait qu'il n'est pas nécessaire de charger en mémoire toutes les données d'entraînement, seulement le sous-ensemble choisi doit être chargé chaque mise à jour. Finalement, la descente de gradient stochastique pousse l'idée de la *mini-batch* à sa limite en ne choisissant qu'un seul exemple aléatoirement à chaque itération.

Analogie

L'intuition de base derrière la descente de gradient peut être illustrée par ce scénario hypothétique.

Une personne est coincée en montagne et essaie d'en descendre (c'est-à-dire essaie de minimiser la hauteur à laquelle elle se trouve). Il y a un épais brouillard tel que la visibilité est extrêmement faible. Par conséquent, le chemin qui descend de la montagne n'est pas facilement visible, alors la personne ne peut utiliser que de l'information locale pour choisir la direction à prendre.

Elle peut utiliser la méthode de descente de gradient, qui consiste à regarder la pente de la colline à sa position actuelle et se déplacer dans le sens où la pente est la plus négative (descente). En utilisant cette méthode, elle finirait par trouver son chemin au bas de la montagne. Cependant, supposons également que la pente de la colline n'est pas immédiatement évidente avec de simples observations et qu'elle nécessite un instrument sophistiqué pour la mesurer. Mesurer la pente de la colline avec l'instrument requiert un certains temps, donc cette personne devrait réduire son utilisation de l'instrument car elle veut descendre le plus rapidement possible. La difficulté est alors de choisir la fréquence à laquelle elle doit mesurer la pente de la colline sans aller hors piste.

Dans cette analogie, l'instrument représente l'algorithme de *rétropropagation du gradient* et la position de la personne sur la montagne représente les paramètres du réseau. La pente de la colline représente la pente de la surface d'erreur à ce point. La distance parcourue entre les mesures (qui est également proportionnelle à la fréquence où elle prend des mesures de la pente) est le taux d'apprentissage.

Règles de mise à jour

Pour élaborer sur les règles de mise à jour des paramètres θ (Étape 3 de l'énumération précédente dans la sous-section 1.3.1), voici la règle de base pour l'algorithme de descente du gradient (Équation 1.3).

$$\theta := \theta - \alpha \nabla_\theta l \tag{1.3}$$

Il existe aussi d'autres méthodes à base de gradient plus avancées telles qu'AdaGrad (Équations 1.4) et AdaDelta (Équations 1.5). Pour alléger la notation, tous les opérateurs de cette sous section tels que $/$, $\sqrt{}$ et ^{2} sont des opérateurs "élément par élément".

AdaGrad [John Duchi, 2010] adapte le taux d'apprentissage automatiquement (Équa-

tion 1.4b) en se basant sur l'accumulation des gradients passés au carré $\boldsymbol{a}$ (Équation 1.4a), initialisée à 0. Ici, ϵ est un chiffre près de zéro, généralement 10^{-6}.

$$\boldsymbol{a} := \boldsymbol{a} + (\nabla_\theta l)^2 \tag{1.4a}$$

$$\tilde{\alpha} = \frac{\alpha}{\sqrt{\boldsymbol{a} + \epsilon}} \tag{1.4b}$$

$$\theta := \theta - \tilde{\alpha} \nabla_\theta l \tag{1.4c}$$

AdaDelta [Zeiler, 2012] est plus complexe. Le taux d'apprentissage est généré dynamiquement au lieu d'être simplement adapté (Équation 1.5b). L'accumulation des anciens gradients au carré est aussi considérée (Équation 1.5a), mais cette fois elle est pondérée par un facteur de détérioration ρ. De plus, un facteur de vitesse $\overline{\Delta\theta}$ est utilisé (Équation 1.5c). De la même façon, il est initialisé à 0 et pondéré par ρ.

$$\boldsymbol{a} := \rho \boldsymbol{a} + (1 - \rho)(\nabla_\theta l)^2 \tag{1.5a}$$

$$\tilde{\alpha} = \frac{\sqrt{\overline{\Delta\theta} + \epsilon}}{\sqrt{\boldsymbol{a} + \epsilon}} \tag{1.5b}$$

$$\overline{\Delta\theta} := \rho \overline{\Delta\theta} + (1 - \rho)(\tilde{\alpha} \nabla_\theta l)^2 \tag{1.5c}$$

$$\theta := \theta - \tilde{\alpha} \nabla_\theta l \tag{1.5d}$$

Contrairement à AdaGrad où le taux d'apprentissage $\tilde{\alpha}$ ne peut que descendre, AdaDelta peut augmenter le $\tilde{\alpha}$, ce qui permet de sortir plus rapidement des plateaux.

Chapitre 2

Précurseurs

Ce chapitre décrit sommairement les principaux algorithmes d'estimation de distribution. Tous sont adaptés au contexte de données binaires. La compréhension de ces algorithmes est pertinente, car la méthode présentée dans ce mémoire se compare à chacun de ceux-ci.

2.1 RBM

Une *Restricted Boltzmann Machine* (RBM), basée sur un champ aléatoire de Markov, est un réseau de neurones artificiel stochastique conçu pour apprendre une distribution de probabilité. Les RBM ont d'abord été inventées sous le nom Harmonium par Paul Smolensky [1986]. Cet algorithme est cependant devenu populaire seulement qu'en 2002, après que Geoffrey Hinton eut inventé un algorithme d'apprentissage rapide pour les entraîner : la *contrastive divergence* [Hinton, 2002]. Les RBM ont trouvé applications dans la réduction de dimensionnalité, la classification, le *feature learning* et le *topic modelling*. L'entrainement de ce modèle peut s'effectuer de façon supervisée ou non, selon la tâche.

Comme son nom l'indique, la RBM est une variante de la *Boltzmann Machine*, avec la restriction que ses unités doivent former un graphe biparti : il existe deux groupes où chaque unité est seulement connectée à toutes les unités du second. Ces groupes d'unités

sont communément dénommés unités visibles et cachées (Figure 2.1). En revanche, les *Boltzmann Machines* traditionnelles sont totalement interconnectées.

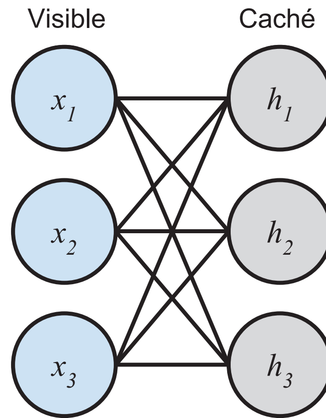


Figure 2.1 – *Restricted Boltzmann Machine* avec 3 unités visibles x_i et 3 unités cachées h_i.

Sous le modèle de la RBM, la probabilité de $\boldsymbol{x}$ est calculée à l'aide de l'équation 2.1, où E est la fonction d'énergie décrite par l'équation 2.2, Z est la fonction de partition qui s'assure que $p(\boldsymbol{x})$ est bien une distribution valide qui somme à 1. $\boldsymbol{x}$ et $\boldsymbol{h}$ représentent les unités, $\boldsymbol{b}$ et $\boldsymbol{c}$ les biais, visibles et cachés respectivement. Comme toujours, $\boldsymbol{W}$ représente les connections entre les unités.

$$p(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{h}} \exp(-E(\boldsymbol{x}, \boldsymbol{h}))}{Z} \tag{2.1}$$

$$E(\boldsymbol{x}, \boldsymbol{h}) = -\boldsymbol{h}^\top \boldsymbol{W} \boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x} - \boldsymbol{c}^\top \boldsymbol{h} \tag{2.2}$$

Lorsque la quantité d'unités cachées est élevée, en générale plus d'une trentaine, la fonction de partition Z devient incalculable. Ce modèle est alors dit intraitable, il peut être résolu en théorie mais prend trop de temps à s'exécuter pour être utilisable en pratique. Donc, dans la majorité des cas, Z doit être estimée pour calculer $p(\boldsymbol{x})$.

2.2 NADE

Presque une décennie passe depuis la RBM et en 2011, Hugo Larochelle et Iain Murray introduisent le modèle *Neural Autoregressive Distribution Estimator* (NADE) Hugo Larochelle [2011], inspiré de la RBM qui a été démontrée comme un modèle puissant pour estimer des distributions discrètes de haute dimension. Bien qu'une RBM ne fournit généralement pas un estimateur de distribution traitable, NADE contourne cette difficulté en décomposant la probabilité jointe des observations en probabilités conditionnelles où chaque conditionnelle est traitable et est modélisée à l'aide d'une fonction non linéaire.

Ce modèle peut être interprété comme un autoencodeur câblé de telle sorte que sa sortie puisse être utilisée pour assigner des probabilités valides aux observations. Précisément, chacune des reconstructions $\widehat{x}_i$ est aussi la probabilité conditionnelle de la $i$ème entrée par rapport aux entrées précédentes, $\widehat{x}_i = p(x_i{=}1 \mid \boldsymbol{x}_{<i})$. Ici $\boldsymbol{x}_{<i}$ implique un ordonnancement des entrées où la $j$ième entrée précède la $i$ème. Par exemple, le cas ou $i{=}3$ donne $\widehat{x}_3 = p(x_3{=}1 \mid x_2, x_1)$. NADE peut apprendre les conditionnelles des entrées dans n'importe quel ordre qui est déterminé avant l'entraînement. Typiquement cet ordre est l'ordre naturel des données (ie : $x_1, x_2, x_3 ... x_D$). Une fois tous les $\widehat{x}_i$ calculés, la probabilité d'un exemple $\boldsymbol{x}$, composée de D éléments, peut être évaluée à l'aide de la règle de multiplication (Équation 2.3).

$$p(\boldsymbol{x}) = p(x_1) \prod_{i=2}^{D} p(x_i \mid \boldsymbol{x}_{<i}) \tag{2.3}$$

La structure utilisée par NADE pour calculer ces probabilités conditionnelles est bien particulière. Comme l'illustre la figure 2.2, la couche d'entrée n'est pas complètement connectée à la couche cachée et certains poids sont liés. Dans cette image, les groupes de poids de même couleur sont dits liés, ce qui implique que les mêmes poids sont réutilisés. Par exemple, les trois poids (en rouge) utilisés pour connecter x_2 au troisième groupe de neurones cachés sont les mêmes que ceux qui sont utilisés pour le lier au quatrième groupe. Cependant, les neurones cachés et de sortie sont des neurones habituels (Figure 1.3). Ceux de sortie utilisent la fonction sigmoïde comme fonction d'activation pour s'assurer que la reconstruction soit entre 0 et 1, et puisse être interprétable comme une probabilité.

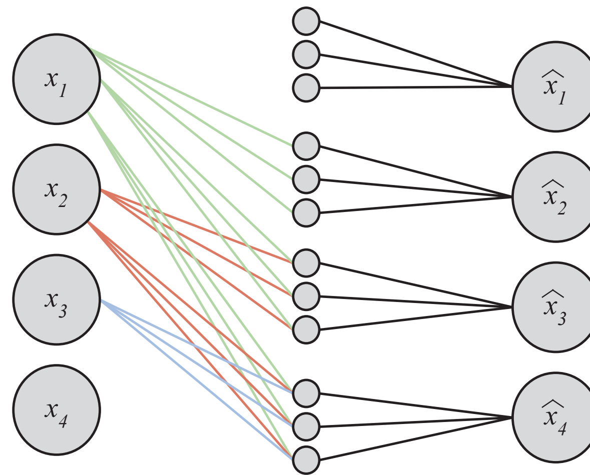


Figure 2.2 – NADE avec 4 entrées et 3 neurones cachés (répété pour chaque entrée). Les groupes de poids de même couleur sont dits liés, ce qui implique que les mêmes poids sont réutilisés (les liens en noir ne sont pas liés). Les biais sont omis du diagramme par soucis de clarté.

L'entrainement de NADE, comme la plupart des modèles qui essaient de reconstruire ses entrées, est effectué à l'aide de la descente de gradient et la fonction objectif de l'entropie croisée que voici (Équation 2.4).

$$l = \sum_{i=1}^{D} -x_i \log \widehat{x}_d - (1 - x_i) \log(1 - \widehat{x}_d) \tag{2.4}$$

2.3 Deep NADE & Orderless NADE

Trois ans plus tard, en 2014, Benigno Uria, Iain Murray et Hugo Larochelle ont amélioré les capacités de NADE en l'étendant à une architecture multicouche et en développant une méthode d'entraînement permettant d'entraîner efficacement une multitude d'ordonnancements des entrées Benigno Uria [2014].

Deep NADE, la version multicouche (Figure 2.3), est une extension simple de NADE. Les couches supplémentaires sont simplement ajoutées comme dans un perceptron multicouche. Malheureusement, l'ajout de ces couches rend l'entraînement beaucoup plus lent, ce qui

rend l'idée pratiquement inutilisable. La complexité algorithmique pour évaluer une entrée augmente de $O(DK)$ pour NADE à $O(DK^2)$ pour la version multicouche, où K est le nombre de neurones dans chaque couche cachée. C'est la seconde innovation qui permet d'exploiter cette version multicouche.

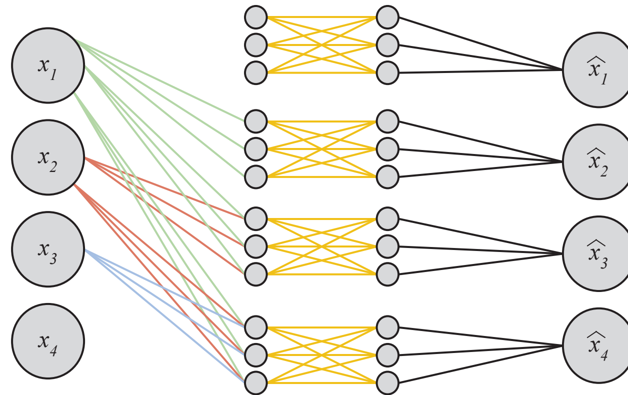
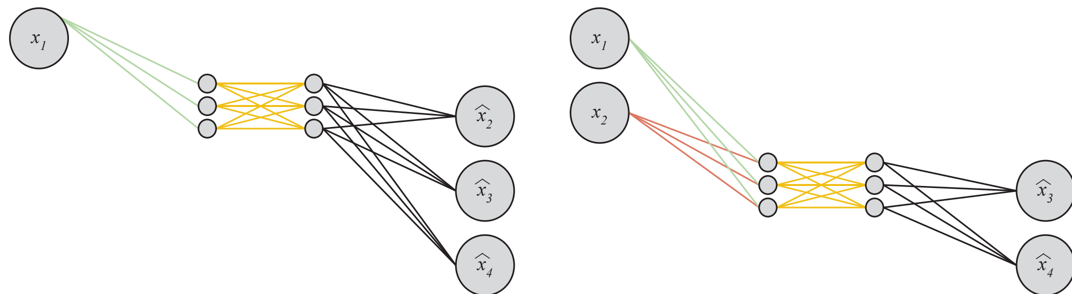


Figure 2.3 – Deep NADE avec 4 entrées, 3 neurones par couche cachée et 2 couches cachées. Les groupes de poids de même couleur sont dits liés (les liens en noir ne sont pas liés). Les biais sont omis du diagramme par soucis de clarté.

L'innovation principale, Orderless NADE, est une nouvelle technique d'entraînement qui permet d'entraîner un NADE de telle façon qu'un ensemble de NADE est simulé. C'est-à-dire que ce NADE peut être considéré comme étant plusieurs NADE ayant tous un ordonnancement différent pour les données d'entrée. De plus, cette technique rend aussi l'entraînement beaucoup plus performant en n'utilisant qu'une sous-partie du réseau à la fois. Contrairement à NADE, qui apprend simplement un ordre fixe des probabilités conditionnelles, Orderless NADE peut apprendre de multiples ordres, ce qui permet de mieux estimer la distribution. Ce travail présente une procédure pour entraîner de façon simultanée un modèle NADE pour chaque ordonnancement possible des entrées, en partageant les paramètres sur tous ces modèles.

Les figures 2.4a et 2.4b sont des exemples de ces entraînements partiels qui apprennent plusieurs ordonnancements de façon simultanée.

Figure 2.4 – Exemples d'entraînements partiels pour Orderless NADE.



(a) Ici les valeurs pour x_1 sont apprises pour les ordonnancements $x_1 x_2 x_? x_?$, $x_1 x_3 x_? x_?$, $x_1 x_4 x_? x_?$.

(b) Ici les valeurs pour x_2 sont apprises pour les ordonnancements $x_1 x_2 x_3 x_?$, $x_1 x_3 x_4 x_?$.

2.4 DARN

Plus tard la même année, le groupe de Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell et Daan Wierstra introduisent le modèle *Deep AutoRegressive Networks* (DARN) Karol Gregor [2014]. Comme NADE et Deep/Orderless NADE, DARN peut être vu comme un autoencodeur génératif qui tente de reconstruire ses entrées. Cependant, contrairement à ces derniers, DARN est intraitable car ses neurones sont stochastiques. Ce modèle multicouche peut être échantillonné par échantillonnage ancestral [Bishop, 2006, Chapter 8.2.1] en effectuant une propagation avant pour chaque entrée, en se basant sur les sorties de la propagation précédente.

Chapter 3

Masked Autoencoder for Distribution Estimation

Ce chapitre décrit en détails le nouveau modèle, *Masked Autoencoder for Distribution Estimation* (MADE). Mathieu Germain a implémenté le modèle, exécuté les expériences et fait évoluer l'idée avec Hugo Larochelle qui a eu l'idée originale. Karol Gregor et Ian Murray ont aussi contribué à l'inspiration originale et à l'écriture de l'article. Cet article a été accepté à la conférence *International Conference on Machine Learning* (ICML) 2015.

3.1 Abstract

There has been a lot of recent interest in designing neural network models to estimate a distribution from a set of examples. We introduce a simple modification for autoencoder neural networks that yields powerful generative models. Our method masks the autoencoder's parameters to respect autoregressive constraints: each input is reconstructed only from previous inputs in a given ordering. Constrained this way, the autoencoder outputs can be interpreted as a set of conditional probabilities, and their product, the full joint probability. We can also train a single network that can decompose the joint probability in multiple different orderings. Our simple framework can be applied to multiple architectures, including

deep ones. Vectorized implementations, such as on GPUs, are simple and fast. Experiments demonstrate that this approach is competitive with state-of-the-art tractable distribution estimators. At test time, the method is significantly faster and scales better than other autoregressive estimators.

3.2 Introduction

Distribution estimation is the task of estimating a joint distribution $p(\mathbf{x})$ from a set of examples $\{\mathbf{x}^{(t)}\}_{t=1}^{T}$, which is by definition a general problem. Many tasks in machine learning can be formulated as learning only specific properties of a joint distribution. Thus a good distribution estimator can be used in many scenarios, including classification Tanya Schmah [2009], denoising or missing input imputation Hoifung Poon [2011]; Laurent Dinh [2014], data (e.g. speech) synthesis Benigno Uria [2015] and many others. The very nature of distribution estimation also makes it a particular challenge for machine learning. In essence, the curse of dimensionality has a distinct impact because, as the number of dimensions of the input space of $\mathbf{x}$ grows, the volume of space in which the model must provide a good answer for $p(\mathbf{x})$ exponentially increases.

Fortunately, recent research has made substantial progress on this task. Specifically, learning algorithms for a variety of neural network models have been proposed Yoshua Bengio [2000]; Hugo Larochelle [2011]; Karol Gregor [2011]; Benigno Uria [2013, 2014]; Diederik P. Kingma [2014]; Danilo Jimenez Rezende [2014]; Yoshua Bengio [2014]; Karol Gregor [2014]; Ian Goodfellow [2014]; Laurent Dinh [2014]. These algorithms are showing great potential in scaling to high-dimensional distribution estimation problems. In this work, we focus our attention on *autoregressive* models (Section 3.4). Computing $p(\mathbf{x})$ exactly for a test example $\mathbf{x}$ is tractable with these models. However, the computational cost of this operation is still larger than typical neural network predictions for a D-dimensional input. For previous deep autoregressive models, evaluating $p(\mathbf{x})$ costs $O(D)$ times more than a simple neural network point predictor.

This paper's contribution is to describe and explore a simple way of adapting autoencoder neural networks that makes them competitive tractable distribution estimators that are faster

than existing alternatives. We show how to mask the weighted connections of a standard autoencoder to convert it into a distribution estimator. The key is to use masks that are designed in such a way that the output is autoregressive for a given ordering of the inputs, i.e. that each input dimension is reconstructed solely from the dimensions preceding it in the ordering. The resulting Masked Autoencoder Distribution Estimator (MADE) preserves the efficiency of a single pass through a regular autoencoder. Implementation on a GPU is straightforward, making the method scalable.

The single hidden layer version of MADE corresponds to the previously proposed autoregressive neural network of Yoshua Bengio [2000]. Here, we go further by exploring deep variants of the model. We also explore training MADE to work simultaneously with multiple orderings of the input observations and hidden layer connectivity structures. We test these extensions across a range of binary datasets with hundreds of dimensions, and compare its statistical performance and scaling to comparable methods.

3.3 Autoencoders

A brief description of the basic autoencoder, on which this work builds upon, is required to clearly grasp what follows. In this paper, we assume that we are given a training set of examples $\{\mathbf{x}^{(t)}\}_{t=1}^{T}$. We concentrate on the case of binary observations, where for every D-dimensional input $\mathbf{x}$, each input dimension x_d belongs in $\{0, 1\}$. The motivation is to learn hidden representations of the inputs that reveal the statistical structure of the distribution that generated them.

An autoencoder attempts to learn a feed-forward, hidden representation $\mathbf{h}(\mathbf{x})$ of its input $\mathbf{x}$ such that, from it, we can obtain a reconstruction $\widehat{\mathbf{x}}$ which is as close as possible to $\mathbf{x}$. Specifically, we have

$$\begin{aligned} \mathbf{h}(\mathbf{x}) &= \mathbf{g}(\mathbf{b} + \mathbf{W}\mathbf{x}) && (3.1) \\ \widehat{\mathbf{x}} &= \mathrm{sigm}(\mathbf{c} + \mathbf{V}\mathbf{h}(\mathbf{x}))\,, && (3.2) \end{aligned}$$

where $\mathbf{W}$ and $\mathbf{V}$ are matrices, $\mathbf{b}$ and $\mathbf{c}$ are vectors, $\mathbf{g}$ is a nonlinear activation function and

$\mathrm{sigm}(a) = 1/(1+\exp(-a))$. Thus, $\mathbf{W}$ represents the connections from the input to the hidden layer, and $\mathbf{V}$ represents the connections from the hidden to the output layer.

To train the autoencoder, we must first specify a training loss function. For binary observations, a natural choice is the cross-entropy loss:

$$\ell(\mathbf{x}) = \sum_{d=1}^{D} -x_d \log \widehat{x}_d - (1-x_d) \log(1-\widehat{x}_d) \ . \tag{3.3}$$

By treating $\widehat{x}_d$ as the model's probability that x_d is 1, the cross-entropy can be understood as taking the form of a negative log-likelihood function. Training the autoencoder corresponds to optimizing the parameters $\{\mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}\}$ to reduce the average loss on the training examples, usually with (mini-batch) stochastic gradient descent.

One advantage of the autoencoder paradigm is its flexibility. In particular, it is straightforward to obtain a deep autoencoder by inserting more hidden layers between the input and output layers. Its main disadvantage is that the representation it learns can be trivial. For instance, if the hidden layer is at least as large as the input, hidden units can each learn to "copy" a single input dimension, so as to reconstruct all inputs perfectly at the output layer. One obvious consequence of this observation is that the loss function of Equation 3.3 isn't in fact a proper log-likelihood function. Indeed, since perfect reconstruction could be achieved, the implied data 'distribution' $q(\mathbf{x}) = \prod_d \widehat{x}_d^{x_d}(1-\widehat{x}_d)^{1-x_d}$ could be learned to be 1 for any $\mathbf{x}$ and thus not be properly normalized ($\sum_{\mathbf{x}} q(\mathbf{x}) \neq 1$).

3.4 Distribution Estimation as Autoregression

An interesting question is what property we could impose on the autoencoder, such that its output can be used to obtain valid probabilities. Specifically, we'd like to be able to write $p(\mathbf{x})$ in such a way that it could be computed based on the output of a properly corrected autoencoder.

First, we can use the fact that, for any distribution, the probability product rule implies that we can always decompose it into the product of its nested conditionals

$$p(\mathbf{x}) = \prod_{d=1}^{D} p(x_d \,|\, \mathbf{x}_{<d}), \quad \text{where } \mathbf{x}_{<d} = [x_1, \ldots, x_{d-1}]^\top. \tag{3.4}$$

By defining $p(x_d\!=\!1 \,|\, \mathbf{x}_{<d}) = \hat{x}_d$, and thus $p(x_d\!=\!0 \,|\, \mathbf{x}_{<d}) = 1\!-\!\hat{x}_d$, the loss of Equation 3.3 becomes a valid negative log-likelihood:

$$\begin{aligned}
-\log p(\mathbf{x}) &= \sum_{d=1}^{D} -\log p(x_d \,|\, \mathbf{x}_{<d}) \\
&= \sum_{d=1}^{D} -x_d \log p(x_d\!=\!1 \,|\, \mathbf{x}_{<d}) \\
&\qquad - (1\!-\!x_d) \log p(x_d\!=\!0 \,|\, \mathbf{x}_{<d}) \\
&= \ell(\mathbf{x})\,.
\end{aligned} \tag{3.5}$$

This connection provides a way to define autoencoders that can be used for distribution estimation. Each output $\widehat{x}_d\!=\!p(x_d \,|\, \mathbf{x}_{<d})$ must be a function taking as input $\mathbf{x}_{<d}$ only and outputting the probability of observing value x_d at the dth dimension. In particular, the autoencoder forms a proper distribution if each output unit $\hat{x}_d$ only depends on the previous input units $\mathbf{x}_{<d}$, and not the other units $\mathbf{x}_{\geq d} = [x_d, \ldots, x_D]^\top$.

We refer to this property as the *autoregressive property*, because computing the negative log-likelihood (3.5) is equivalent to sequentially predicting (regressing) each dimension of input $\mathbf{x}$.

3.5 Masked Autoencoders

The question now is how to modify the autoencoder so as to satisfy the autoregressive property. Since output $\hat{x}_d$ must depend only on the preceding inputs $\mathbf{x}_{<d}$, it means that there must be no computational path between output unit $\hat{x}_d$ and any of the input units $x_d, \ldots, x_D$. In other words, for each of these paths, at least one connection (in matrix $\mathbf{W}$ or $\mathbf{V}$) must be 0.

A convenient way of zeroing connections is to elementwise-multiply each matrix by a binary mask matrix, whose entries that are set to 0 correspond to the connections we wish to remove. For a single hidden layer autoencoder, we write

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{b} + (\mathbf{W} \odot \mathbf{M}^{\mathbf{W}})\mathbf{x}) \tag{3.6}$$

$$\hat{\mathbf{x}} = \mathrm{sigm}(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}(\mathbf{x})) \tag{3.7}$$

where $\mathbf{M}^{\mathbf{W}}$ and $\mathbf{M}^{\mathbf{V}}$ are the masks for $\mathbf{W}$ and $\mathbf{V}$ respectively. It is thus left to the masks $\mathbf{M}^{\mathbf{W}}$ and $\mathbf{M}^{\mathbf{V}}$ to satisfy the autoregressive property.

To impose the autoregressive property we first assign each unit in the hidden layer an integer m between 1 and $D-1$ inclusively. The kth hidden unit's number $m(k)$ gives the maximum number of input units to which it can be connected. We disallow $m(k)=D$ since this hidden unit would depend on all inputs and could not be used in modelling any of the conditionals $p(x_d \mid \mathbf{x}_{<d})$. Similarly, we exclude $m(k)=0$, as it would create constant hidden units.

The constraints on the maximum number of inputs to each hidden unit are encoded in the matrix masking the connections between the input and hidden units:

$$M^{\mathbf{W}}_{k,d} = 1_{m(k)\geq d} = \begin{cases} 1 & \text{if } m(k) \geq d \\ 0 & \text{otherwise,} \end{cases} \tag{3.8}$$

for $d \in \{1, \ldots, D\}$ and $k \in \{1, \ldots, K\}$. Overall, we need to encode the constraint that the dth output unit is only connected to $\mathbf{x}_{<d}$ (and thus not to $\mathbf{x}_{\geq d}$). Therefore the output weights can only connect the dth output to hidden units with $m(k)<d$, i.e. units that are connected to at most $d-1$ input units. These constraints are encoded in the output mask matrix:

$$M^{\mathbf{V}}_{d,k} = 1_{d>m(k)} = \begin{cases} 1 & \text{if } d > m(k) \\ 0 & \text{otherwise,} \end{cases} \tag{3.9}$$

again for $d \in \{1, \ldots, D\}$ and $k \in \{1, \ldots, K\}$. Notice that, from this rule, no hidden units will be connected to the first output unit $\hat{x}_1$, as desired.

From these mask constructions, we can easily demonstrate that the corresponding masked autoencoder satisfies the autoregressive property. First, we note that, since the masks $\mathbf{M}^{\mathbf{V}}$

and $\mathbf{M}^{\mathbf{W}}$ represent the network's connectivity, their matrix product $\mathbf{M}^{\mathbf{V},\mathbf{W}} = \mathbf{M}^{\mathbf{V}}\mathbf{M}^{\mathbf{W}}$ represents the connectivity between the input and the output layer. Specifically, $M^{\mathbf{V},\mathbf{W}}_{d',d}$ is the number of network paths between output unit $\hat{x}_{d'}$ and input unit x_d. Thus, to demonstrate the autoregressive property, we need to show that $\mathbf{M}^{\mathbf{V},\mathbf{W}}$ is strictly lower diagonal, i.e. $M^{\mathbf{V},\mathbf{W}}_{d',d}$ is 0 if $d' \leq d$. By definition of the matrix product, we have:

$$M^{\mathbf{V},\mathbf{W}}_{d',d} = \sum_{k=1}^{K} M^{\mathbf{V}}_{d',k} M^{\mathbf{W}}_{k,d} = \sum_{k=1}^{K} 1_{d'>m(k)} 1_{m(k)\geq d}. \tag{3.10}$$

If $d' \leq d$, then there are no values for $m(k)$ such that it is both strictly less than d' and greater or equal to d. Thus $M^{\mathbf{V},\mathbf{W}}_{d',d}$ is indeed 0.

Constructing the masks $\mathbf{M}^{\mathbf{V}}$ and $\mathbf{M}^{\mathbf{W}}$ only requires an assignment of the $m(k)$ values to each hidden unit. One could imagine trying to assign an (approximately) equal number of units to each legal value of $m(k)$. In our experiments, we instead set $m(k)$ by sampling from a uniform discrete distribution defined on integers from 1 to $D-1$, independently for each of the K hidden units.

Previous work on autoregressive neural networks have also found it advantageous to use direct connections between the input and output layers Yoshua Bengio [2000]. In this context, the reconstruction becomes:

$$\hat{\mathbf{x}} = \mathrm{sigm}(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}(\mathbf{x}) + (\mathbf{A} \odot \mathbf{M}^{\mathbf{A}})\mathbf{x}), \tag{3.11}$$

where $\mathbf{A}$ is the parameter connection matrix and $\mathbf{M}^{\mathbf{A}}$ is its mask matrix. To satisfy the autoregressive property, $\mathbf{M}^{\mathbf{A}}$ simply needs to be a strictly lower diagonal matrix, filled otherwise with ones. We used such direct connections in our experiments as well.

3.5.1 Deep MADE

One advantage of the masked autoencoder framework described in the previous section is that it naturally generalizes to deep architectures. Indeed, as we'll see, by assigning a maximum number of connected inputs to all units across the deep network, masks can be similarly constructed so as to satisfy the autoregressive property.

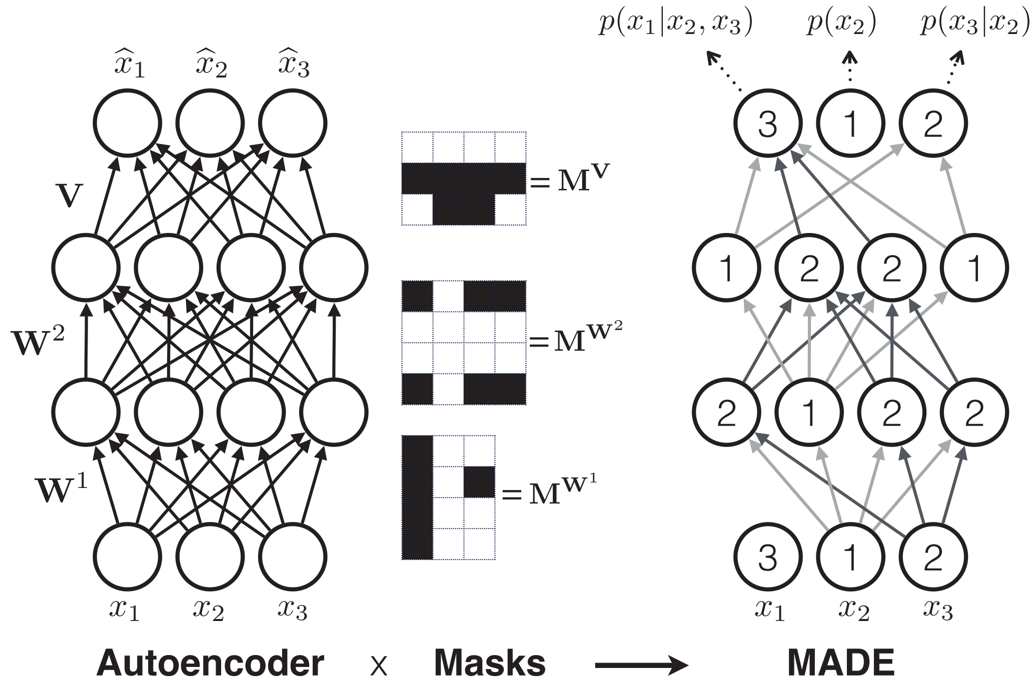


Figure 3.1 – **Left: Conventional three hidden layer autoencoder**. Input in the bottom is passed through fully connected layers and point-wise nonlinearities. In the final top layer, a reconstruction specified as a probability distribution over inputs is produced. As this distribution depends on the input itself, a standard autoencoder cannot predict or sample new data. **Right: MADE**. The network has the same structure as the autoencoder, but a set of connections is removed such that each input unit is only predicted from the previous ones, using multiplicative binary masks ($\mathbf{M}^{\mathbf{W}^1}$, $\mathbf{M}^{\mathbf{W}^2}$, $\mathbf{M}^{\mathbf{V}}$). In this example, the ordering of the input is changed from 1,2,3 to 3,1,2. This change is explained in section 3.5.2, but is not necessary for understanding the basic principle. The numbers in the hidden units indicate the maximum number of inputs on which the kth unit of layer l depends. The masks are constructed based on these numbers (see Equations 3.12 and 3.13). These masks ensure that MADE satisfies the autoregressive property, allowing it to form a probabilistic model, in this example $p(\mathbf{x}) = p(x_2)\, p(x_3|x_2)\, p(x_1|x_2, x_3)$. Connections in light gray correspond to paths that depend only on 1 input, while the dark gray connections depend on 2 inputs.

For networks with $L > 1$ hidden layers, we use superscripts to index the layers. The first hidden layer matrix (previously $\mathbf{W}$) will be denoted $\mathbf{W}^1$, the second hidden layer matrix will be $\mathbf{W}^2$, and so on. The number of hidden units (previously K) in each hidden layer will be similarly indexed as K^l, where l is the hidden layer index. We will also generalize the notation for the maximum number of connected inputs of the kth unit in the lth layer to

$m^l(k)$.

We've already discussed how to define the first layer's mask matrix such that it ensures that its kth unit is connected to at most $m(k)$ (now $m^1(k)$) inputs. To impose the same property on the second hidden layer, we must simply make sure that each unit k' is only connected to first layer units connected to at most $m^2(k')$ inputs, i.e. the first layer units such that $m^1(k) \leq m^2(k')$.

One can generalize this rule to any layer l, as follows:

$$M^{\mathbf{W}^l}_{k',k} = 1_{m^l(k') \geq m^{l-1}(k)} = \begin{cases} 1 & \text{if } m^l(k') \geq m^{l-1}(k) \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

Also, taking $l = 0$ to mean the input layer and defining $m^0(d) = d$ (which is intuitive, since the dth input unit indeed takes its values from the d first inputs), this definition also applies for the first hidden layer weights. As for the output mask, we simply need to adapt its definition by using the connectivity constraints of the last hidden layer $m^L(k)$ instead of the first:

$$M^{\mathbf{V}}_{d,k} = 1_{d > m^L(k)} = \begin{cases} 1 & \text{if } d > m^L(k) \\ 0 & \text{otherwise.} \end{cases} \tag{3.13}$$

Like for the single hidden layer case, the values for $m^l(k)$ for each hidden layer $l \in \{1, \ldots, L\}$ are sampled uniformly. To avoid unconnected units, the value for $m^l(k)$ is sampled to be greater than or equal to the minimum connectivity at the previous layer, i.e. $\min_{k'} m^{l-1}(k')$.

3.5.2 Order-agnostic training

So far, we've assumed that the conditionals modelled by MADE were consistent with the natural ordering of the dimensions of $\mathbf{x}$. However, we might be interested in modelling the conditionals associated with an arbitrary ordering of the input's dimensions.

Specifically, Benigno Uria [2014] have shown that training an autoregressive model on *all* orderings can be beneficial. We refer to this approach as order-agnostic training. It can be achieved by sampling an ordering before each stochastic/minibatch gradient update of

the model. There are two advantages of this approach. Firstly, missing values in partially observed input vectors can be imputed efficiently: we invoke an ordering where observed dimensions are all before unobserved ones, making inference straightforward. Secondly, an ensemble of autoregressive models can be constructed on the fly, by exploiting the fact that the conditionals for two different orderings are not guaranteed to be exactly consistent (and thus technically correspond to slightly different models). An ensemble is then easily obtained by sampling a set of orderings, computing the probability of $\mathbf{x}$ under each ordering and averaging.

Conveniently, in MADE, the ordering is simply represented by the vector $^0 = [m^0(1), \ldots,$ $m^0(D)]$. Specifically, $m^0(d)$ corresponds to the position of the original d^{th} dimension of $\mathbf{x}$ in the product of conditionals. Thus, a random ordering can be obtained by randomly permuting the ordered vector $[1, \ldots, D]$. From these values of each ^{0}, the first hidden layer mask matrix can then be created. During order-agnostic training, randomly permuting the last value of ^{0} again is sufficient to obtain a new random ordering.

3.5.3 Connectivity-agnostic training

One advantage of order-agnostic training is that it effectively allows us to train as many models as there are orderings, using a common set of parameters. This can be exploited by creating ensembles of models at test time.

In MADE, in addition to choosing an ordering, we also have to choose each hidden unit's connectivity constraint $m^l(k)$. Thus, we could imaging training MADE to also be agnostic of the connectivity pattern generated by these constraints. To achieve this, instead of sampling the values of $m^l(k)$ for all units and layers once and for all before training, we actually resample them for each training example or minibatch. This is still practical, since the operation of creating the masks is easy to parallelize. Denoting $^l = [m^l(1), \ldots, m^l(K^l)]$, and assuming an element-wise and parallel implementation of the operation $1_{a \geq b}$ for vectors, such that $\mathbf{1_{a \geq b}}$ is a matrix whose i, j element is $1_{a_i \geq b_j}$, then the hidden layer masks are simply $\mathrm{M}^{\mathbf{W}^l} = \mathbf{1}_{^l \geq ^{l-1}}$.

By resampling the connectivity of hidden units for every update, each hidden unit will have

a constantly changing number of incoming inputs during training. However, the absence of a connection is indistinguishable from an instantiated connection to a zero-valued unit, which could confuse the neural network during training. In a similar situation, Benigno Uria [2014] informed each hidden unit which units were providing input with binary indicator variables, connected with additional learnable weights. We considered applying a similar strategy, using companion weight matrices $\mathbf{U}^l$, that are also masked by $\mathbf{M}^{\mathbf{W}^l}$ but connected to a constant one-valued vector:

$$\mathbf{h}^l(\mathbf{x}) = \mathbf{g}(\mathbf{b}^l + (\mathbf{W}^l \odot \mathbf{M}^{\mathbf{W}^l})\mathbf{h}^{l-1}(\mathbf{x}) + (\mathbf{U}^l \odot \mathbf{M}^{\mathbf{W}^l})\mathbf{1}) \tag{3.14}$$

An analogous parametrization of the output layer was also employed. These connectivity conditioning weights were only sometimes useful. In our experiments, we treated the choice of using them as a hyperparameter.

Moreover, we've found in our experiments that sampling masks for every example could sometimes over-regularize MADE and provoke underfitting. To fix this issue, we also considered sampling from only a finite list of masks. During training, MADE cycles through this list, using one for every update. At test time, we then average probabilities obtained for all masks in the list.

Algorithm 1 details how $p(\mathbf{x})$ is computed by MADE, as well as how to obtain the gradient of $\ell(\mathbf{x})$ for stochastic gradient descent training. For simplicity, the pseudocode assumes order-agnostic and connectivity-agnostic training, doesn't assume the use of conditioning weight matrices or of direct input/output connections. Figure 3.1 also illustrates an example of such a two-layer MADE network, along with its $m^l(k)$ values and its masks.

3.6 Related Work

There has been a lot of recent work on exploring the use of feed-forward, autoencoder-like neural networks as probabilistic generative models. Part of the motivation behind this research is to test the common assumption that the use of models with probabilistic latent variables and intractable partition functions (such as the restricted Boltzmann

machine Salakhutdinov and Murray [2008]), is a necessary evil in designing powerful generative models for high-dimensional data.

The work on the neural autoregressive distribution estimator or NADE Hugo Larochelle [2011] has illustrated that feed-forward architectures can in fact be used to form state-of-the-art and even tractable distribution estimators.

Recently, a deep extension of NADE was proposed, improving even further the state-of-the-art in distribution estimation Benigno Uria [2014]. This work introduced a randomized training procedure, which (like MADE) has nearly the same cost per iteration as a standard autoencoder. Unfortunately, deep NADE models still require D feed-forward passes through the network to evaluate the probability $p(\mathbf{x})$ of a D-dimensional test vector. The computation of the first hidden layer's activations can be shared across these passes, although is slower in practice than evaluating a single pass in a standard autoencoder. In deep networks with K hidden units per layer, it costs $O(DK^2)$ to evaluate a test vector.

Deep AutoRegressive Networks [DARN, Karol Gregor, 2014], also provide probabilistic models with roughly the same training costs as standard autoencoders. DARN's latent representation consist of binary, stochastic hidden units. While simulating from these models is fast, evaluation of exact test probabilities requires summing over all configurations of the latent representation, which is exponential in computation. Monte Carlo approximation is thus recommended.

The main advantage of MADE is that evaluating probabilities retains the efficiency of autoencoders, with minor additional cost for simple masking operations. Table 3.1 lists the computational complexity for exact computation of probabilities for various models. DARN and RBMs are exponential in dimensionality of the hiddens or data, whereas NADE and MADE are polynomial. MADE only requires one pass through the autoencoder rather than the D passes required by NADE. In practice, we also observe that the single-layer MADE is an order of magnitude faster than a one-layer NADE, for the same hidden layer size, despite NADE sharing computation to get the same asymptotic scaling. NADE's computations cannot be vectorized as efficiently. The deep versions of MADE also have better scaling than NADE at test time. The training costs for MADE, DARN, and deep NADE will all be similar.

Table 3.1 – Complexity of the different models in Table 3.7, to compute an exact test negative log-likelihood. R is the number of orderings used, D is the input size, and K is the hidden layer size (assuming equally sized hidden layers).

Model	O_{NLL}
RBM 25 CD steps	$O(\min(2^D K,\ D2^K))$
DARN	$O(2^K D)$
NADE (fixed order)	$O(DK)$
EoNADE 1hl, R ord.	$O(RDK)$
EoNADE 2hl, R ord.	$O(RDK^2)$
MADE 1hl, 1 ord.	$O(DK+D^2)$
MADE 2hl, 1 ord.	$O(DK+K^2+D^2)$
MADE 1hl, R ord.	$O(R(DK+D^2))$
MADE 2hl, R ord.	$O(R(DK+K^2+D^2))$

Before the work on NADE, Yoshua Bengio [2000] proposed a neural network architecture that corresponds to the special case of a single hidden layer MADE model, without randomization of input ordering and connectivity. A contribution of our work is to go beyond this special case, exploring deep variants and order/connectivity-agnostic training.

An interesting interpretation of the autoregressive mask sampling is as a structured form of dropout regularization Nitish Srivastava [2014]. Specifically, it bears similarity with the masking in dropconnect networks Li Wan [2013]. The exception is that the masks generated here must guaranty the autoregressive property of the autoencoder, while in Li Wan [2013], each element in the mask is generated independently.

3.7 Experiments

To test the performance of our model we considered two different benchmarks: a suite of UCI binary datasets, and the binarized MNIST dataset. The code [1] is available on GitHub. The results reported here are the average negative log-likelihood on the test set of each respective dataset. All experiments were made using stochastic gradient descent (SGD) with

1. https://github.com/mgermain/MADE/releases/tag/ICML2015

Table 3.2 – Number of input dimensions and numbers of examples in the train, validation, and test splits.

Name	# Inputs	Train	Valid.	Test
Adult	123	5000	1414	26147
Connect4	126	16000	4000	47557
DNA	180	1400	600	1186
Mushrooms	112	2000	500	5624
NIPS-0-12	500	400	100	1240
OCR-letters	128	32152	10000	10000
RCV1	150	40000	10000	150000
Web	300	14000	3188	32561

mini-batches of size 100 and a lookahead of 30 for early stopping.

3.7.1 UCI evaluation suite

We use the binary UCI evaluation suite that was first put together in Hugo Larochelle [2011]. It's a collection of 7 relatively small datasets from the University of California, Irvine machine learning repository and the OCR-letters dataset from the Stanford AI Lab. Table 3.2 gives an overview of the scale of those datasets and the way they were split.

The experiments were run with networks of 500 units per hidden layer, using the adadelta learning update Zeiler [2012] with a decay of 0.95. The other hyperparameters were varied as Table 3.3 indicates. We note as *# of masks* the number of different masks through which MADE cycles during training. In the no limit case, masks are sampled on the fly and never explicitly reused unless re-sampled by chance. In this situation, at validation and test time, 300 and 1000 sampled masks are used for averaging probabilities.

The results are reported in Table 3.4. We see that MADE is among the best performing models on half of the datasets and is competitive otherwise. To reduce clutter, we have not reported standard deviations in the same table as the NLL. However, for completeness we report it in Table 3.5.

An analysis of the hyperparameters selected for each dataset reveals no clear winner. How-

Table 3.3 – UCI Grid Search

Hyperparameter	Values tried
# Hidden Layer	1, 2
Activation function	ReLU, Softplus
Adadelta epsilon	10^{-5}, 10^{-7}, 10^{-9}
Conditioning Weights	True, False
# of orderings	1, 8, 16, 32, No Limit

Table 3.4 – Negative log-likelihood test results of different models on multiple datasets. The best result as well as any other result with an overlapping confidence interval is shown in bold. Note that since the variance of DARN was not available, we considered it to be zero.

Model	Adult	Connect4	DNA	Mushrooms	NIPS-0-12	OCR-letters	RCV1	Web
MoBernoullis	20.44	23.41	98.19	14.46	290.02	40.56	47.59	30.16
RBM	16.26	22.66	96.74	15.15	277.37	43.05	48.88	29.38
FVSBN	**13.17**	12.39	83.64	10.27	276.88	39.30	49.84	29.35
NADE (fixed order)	**13.19**	11.99	84.81	9.81	**273.08**	**27.22**	46.66	28.39
EoNADE 1hl (16 ord.)	**13.19**	12.58	82.31	9.69	**272.39**	**27.32**	**46.12**	**27.87**
DARN	13.19	11.91	81.04	**9.55**	274.68	≈28.17	≈**46.10**	≈28.83
MADE	**13.12**	**11.90**	83.63	9.68	280.25	28.34	47.10	28.53
MADE mask sampling	**13.13**	**11.90**	**79.66**	9.69	277.28	30.04	46.74	**28.25**

ever, we do see from Table 3.4 that when the mask sampling helps, it helps quite a bit and when it does not, the impact is negligible on all but OCR-letters. Another interesting note is that the conditioning weights had almost no influence except on NIPS-0-12 where it helped.

3.7.2 Binarized MNIST evaluation

The version of MNIST we used is the one binarized by Salakhutdinov and Murray [2008]. MNIST is a set of 70,000 hand written digits of 28×28 pixels. We use the same split as in Hugo Larochelle [2011], consisting of 50,000 for the training set, 10,000 for the validation set and 10,000 for the test set.

Experiments were run using the adagrad learning update John Duchi [2010], with an epsilon

Table 3.5 – Negative log-likelihood and 95% confidence intervals for Table 3.4.

	MADE		EoNADE
Dataset	Fixed mask	Mask sampling	16 ord.
Adult	$13.12_{\pm 0.05}$	$13.13_{\pm 0.05}$	$13.19_{\pm 0.04}$
Connect4	$11.90_{\pm 0.01}$	$11.90_{\pm 0.01}$	$12.58_{\pm 0.01}$
DNA	$83.63_{\pm 0.52}$	$79.66_{\pm 0.63}$	$82.31_{\pm 0.46}$
Mushrooms	$9.68_{\pm 0.04}$	$9.69_{\pm 0.03}$	$9.69_{\pm 0.03}$
NIPS-0-12	$280.25_{\pm 1.05}$	$275.92_{\pm 1.01}$	$272.39_{\pm 1.08}$
Ocr-letters	$28.34_{\pm 0.22}$	$30.04_{\pm 0.22}$	$27.32_{\pm 0.19}$
RCV1	$47.10_{\pm 0.11}$	$46.74_{\pm 0.11}$	$46.12_{\pm 0.11}$
Web	$28.53_{\pm 0.20}$	$28.25_{\pm 0.20}$	$27.87_{\pm 0.20}$

Table 3.6 – Binarized MNIST Grid Search

Hyperparameter	Values tried
# Hidden Layer	1, 2
Learning Rate	0.1, 0.05, 0.01, 0.005
# of masks	1, 2, 4, 8, 16, 32, 64

of 10^{-6}. Since MADE is much more efficient than NADE, we considered varying the hidden layer size from 500 to 8000 units. Seeing that increasing the number of units tended to always help, we used 8000. Even with such a large hidden layer, our GPU implementation of MADE was quite efficient. Using a single mask, one training epoch requires about 14 and 44 seconds, for one hidden layer and two hidden layer MADE respectively. Using 32 sampled masks, training time increases to 33 and 100 respectively. These timings are all less than our GPU implementation of the 500 hidden units NADE model, which requires about 130 seconds per epoch. These timings were obtained on a K20 NVIDIA GPU.

Building on what we learned on the UCI experiments, we set the activation function to be ReLU and the conditioning weights were not used. The hyperparameters that were varied are in Table 3.6.

The results are reported in Table 3.7, alongside other results taken from the literature. Again,

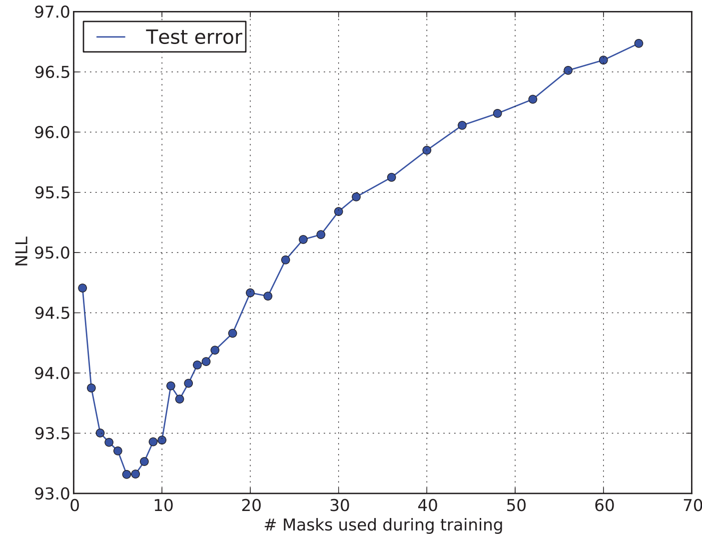


Figure 3.2 – Impact of the number of masks used with a single hidden layer, 500 hidden units network, on binarized MNIST.

we report standard deviations in a seperate table (Table). Despite its tractability, MADE is competitive with other models. Of note is the fact that the best MADE model outperforms the single layer NADE network, which was otherwise the best model among those requiring only a single feed-forward pass to compute log probabilities.

In these experiments, we clearly observed the over-regularization phenomenon from using too many masks. When more than four orderings were used, the deeper variant of MADE always yielded better results. For the two layer model, adding masks during training helped up to 64, at which point the negative log-likelihood started to increase. We observed a similar pattern for the single layer model, but in this case the dip was around 8 masks. Figure 3.2 illustrates this behaviour more precisely for a single layer MADE with 500 hidden units, trained by only varying the number of masks used and the size of the mini-batches (83, 100, 128).

We randomly sampled 100 digits from our best performing model from Table 3.7 and compared them with their nearest neighbor in the training set (Figure 3.3), to ensure that the generated samples are not simple memorization. Each row of digits uses a different mask

Table 3.7 – Negative log-likelihood test results of different models on the binarized MNIST dataset.

Model	$-\log p$	
RBM (500 h, 25 CD steps)	$\approx$ 86.34	Intractable
DBM 2hl	$\approx$ 84.62	
DBN 2hl	$\approx$ 84.55	
DARN n_h=500	$\approx$ 84.71	
DARN n_h=500, adaNoise	$\approx$ 84.13	
MoBernoullis K=10	168.95	Tractable
MoBernoullis K=500	137.64	
NADE 1hl (fixed order)	88.33	
EoNADE 1hl (128 orderings)	87.71	
EoNADE 2hl (128 orderings)	85.10	
MADE 1hl (1 mask)	88.40	
MADE 2hl (1 mask)	89.59	
MADE 1hl (32 masks)	88.04	
MADE 2hl (32 masks)	86.64	

Table 3.8 – Binarized MNIST negative log-likelihood and 95% confidence intervals for Table 3.7.

Model	
MADE 1hl (1 mask)	$88.40_{\pm 0.45}$
MADE 2hl (1 mask)	$89.59_{\pm 0.46}$
MADE 1hl (32 masks)	$88.04_{\pm 0.44}$
MADE 2hl (32 masks)	$86.64_{\pm 0.44}$

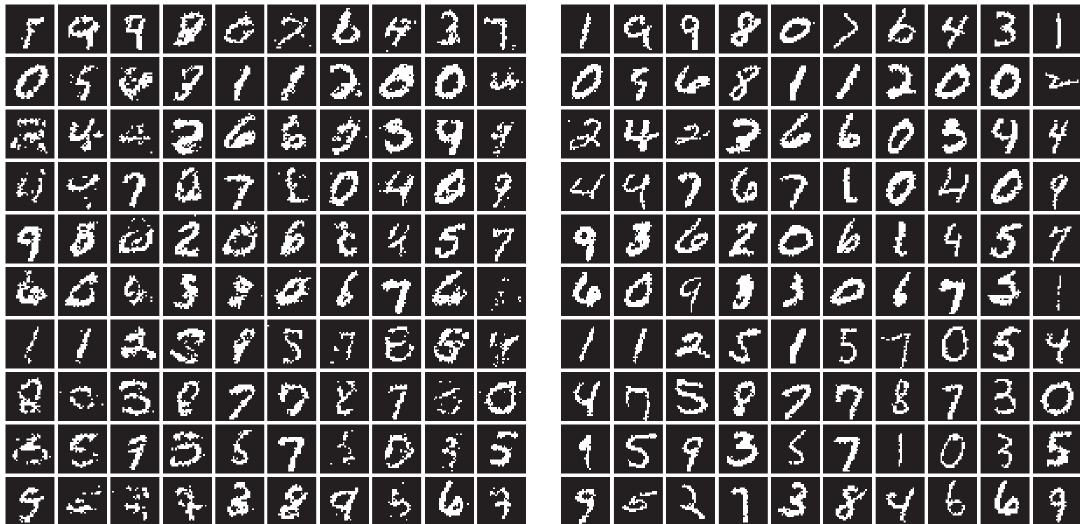


Figure 3.3 – Left: Samples from a 2 hidden layer MADE. Right: Nearest neighbour in binarized MNIST.

that was seen at training time by the network.

3.8 Conclusion

We proposed MADE, a simple modification of autoencoders allowing them to be used as distribution estimators. MADE demonstrates that it is possible to get direct, cheap estimates of high-dimensional joint probabilities, from a single pass through an autoencoder. Like standard autoencoders, our extension is easy to vectorize and implement on GPUs. MADE can evaluate high-dimensional probably distributions with better scaling than before, while maintaining state-of-the-art statistical performance.

Acknowledgments

We thank Marc-Alexandre Côté for helping to implement NADE in Theano and the whole Theano Frédéric Bastien [2012]; James Bergstra [2010] team of contributors. We also thank NSERC, Calcul Québec and Compute Canada.

Algorithm 1 Computation of $p(\mathbf{x})$ and learning gradients for MADE with order and connectivity sampling. D is the size of the input, L the number of hidden layers and K the number of hidden units.

Input: training observation vector $\mathbf{x}$

Output: $p(\mathbf{x})$ and gradients of $-\log p(\mathbf{x})$ on parameters

\# Sampling ^{l} vectors

$^{0} \leftarrow \text{shuffle}([1, \ldots, D])$

for l from 1 to L **do**

  for k from 1 to K^l **do**

    $m^l(k) \leftarrow \text{Uniform}([\min_{k'} m^{l-1}(k'), \ldots, D-1])$

  end for

end for

\# Constructing masks for each layer

for l from 1 to L **do**

  $\mathbf{M}^{\mathbf{W}^l} \leftarrow \mathbf{1}_{^{l} \geq ^{l-1}}$

end for

$\mathbf{M}^{\mathbf{V}} \leftarrow \mathbf{1}_{^{0} > ^{L}}$

\# Computing $p(\mathbf{x})$

$\mathbf{h}^0(\mathbf{x}) \leftarrow \mathbf{x}$

for l from 1 to L **do**

  $\mathbf{h}^l(\mathbf{x}) \leftarrow \mathbf{g}(\mathbf{b}^l + (\mathbf{W}^l \odot \mathbf{M}^{\mathbf{W}^l})\mathbf{h}^{l-1}(\mathbf{x}))$

end for

$\widehat{\mathbf{x}} \leftarrow \text{sigm}(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}^L(\mathbf{x}))$

$p(\mathbf{x}) \leftarrow \exp\left(\sum_{d=1}^{D} x_d \log \widehat{x}_d + (1-x_d)\log(1-\widehat{x}_d)\right)$

\# Computing gradients of $-\log p(\mathbf{x})$

$\text{tmp} \leftarrow \widehat{\mathbf{x}} - \mathbf{x}$

$\delta\mathbf{c} \leftarrow \text{tmp}$

$\delta\mathbf{V} \leftarrow \left(\text{tmp}\; \mathbf{h}^L(\mathbf{x})^\top\right) \odot \mathbf{M}^{\mathbf{V}}$

$\text{tmp} \leftarrow (\text{tmp}^\top(\mathbf{V} \odot \mathbf{M}^{\mathbf{V}}))^\top$

for l from L to 1 **do**

  $\text{tmp} \leftarrow \text{tmp} \odot \mathbf{g}'(\mathbf{b}^l + (\mathbf{W}^l \odot \mathbf{M}^{\mathbf{W}^l})\mathbf{h}^{l-1}(\mathbf{x}))$

  $\delta\mathbf{b}^l \leftarrow \text{tmp}$

  $\delta\mathbf{W}^l \leftarrow \left(\text{tmp}\; \mathbf{h}^{l-1}(\mathbf{x})^\top\right) \odot \mathbf{M}^{\mathbf{W}^l}$

  $\text{tmp} \leftarrow (\text{tmp}^\top(\mathbf{W}^l \odot \mathbf{M}^{\mathbf{W}^l}))^\top$

end for

return $p(\mathbf{x}), \delta\mathbf{b}^1, \ldots, \delta\mathbf{b}^L, \delta\mathbf{W}^1, \ldots, \delta\mathbf{W}^L, \delta\mathbf{c}, \delta\mathbf{V}$

Conclusion

Ce mémoire a introduit MADE, un nouveau modèle génératif simple et rapide permettant d'estimer la distribution de données binaires tout en conservant des résultats du calibre de l'état de l'art. Tout d'abord, les notions de base en apprentissage automatique nécessaire à la compréhension de ce modèle ont été introduites. Un survol des différents modèles de l'état de l'art à été fait pour permettre de mieux comprendre l'impact de MADE. Ce dernier a aussi été validé sur une dizaine d'ensembles de données et comparé avec les meilleurs modèles contemporains du domaine.

Il est important de souligner une autre contribution, plus technique, qui a grandement accéléré l'accomplissement de ce mémoire. La création de l'outil Smart Dispatch [2], qui permet de lancer plusieurs expériences simultanément de facon simple et efficace sur différentes grappes de calcul, a permis le déroulement fluide et rapide de cette recherche et de bien d'autres.

Au final, grâce à la simplicité de MADE, il serait facile de développer de nouveaux modèles fondés sur celui-ci. Lors de futurs travaux de recherche, il serait intéresant d'explorer différentes façons d'améliorer et d'étendre les capacités de ce modèle.

Avant d'explorer les modifications possibles du modèle, plusieurs autres méthodes d'entraînement plus avancées pourraient être testées pour voir comment elles se comportent lorsque plus d'attention est portée à l'entraînement. Des méthodes telles qu'une combinaison de la descente de gradient et *saddle-free Newton* [Yann Dauphin, 2014] pourraient être envisagées.

2. https://github.com/SMART-Lab/smartdispatch

CONCLUSION

Un bon endroit pour débuter serait une analyse détaillée de l'impact que certaines variations du modèle ont sur les performances. Les différentes configurations de masques pourraient avoir un impact non négligeable. Par exemple, déterminer si un masque éparse ou dense est préférable, si les différences entre les masques des différentes couches ont de l'importance. Par rapport aux différents ensembles de données, comprendre comment la performance est affectée par la dimension des entrées, la quantité de données disponibles, etc. En se basant sur cette analyse, il serait possible de développer une nouvelle façon de générer des masques d'une plus grande variété ou simplement de meilleure qualité.

Ensuite, une version de MADE compatible avec des données réelles, tout comme le démontre Benigno Uria [2015], serait d'une très grande utilité car la grande majorité des ensembles de données ne sont pas qu'uniquement binaires.

Un défi considérable serait de développer une version convolutionelle de MADE pour explorer à quel point les performances peuvent être améliorées sur les images.

Finalement, pour aller encore plus loin, introduire un méchanisme d'attention au sein de MADE améliorerait définitivement les performances sur une multitude d'ensembles de données.

Bibliographie

H. L. Benigno Uria, Iain Murray. RNADE : The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems 26 (NIPS 2013)*, pages 2175–2183, 2013.

H. L. Benigno Uria, Iain Murray. A deep and tractable density estimator. In *Proceedings of the 31th International Conference on Machine Learning, (ICML 2014)*, pages 467–475, 2014.

S. R. C. V.-B. Benigno Uria, Iain Murray. Modelling acoustic feature dependencies with artificial neural networks : Trajectory-RNADE. In *Proceedings of the 40th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2015)*. IEEE Signal Processing Society, 2015.

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

D. W. Danilo Jimenez Rezende, Shakir Mohamed. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31th International Conference on Machine Learning (ICML 2014)*, pages 1278–1286, 2014.

M. W. Diederik P. Kingma. Auto-encoding variational bayes. In *Proceedings of the 2nd International Conference on Learning Representations (ICLR 2014)*, 2014.

R. P. J. B.-I. J. G. A. B. N. B. Y. B. Frédéric Bastien, Pascal Lamblin. Theano : new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8) :1771–1800, August 2002. ISSN 0899-7667.

P. D. Hoifung Poon. Sum-product networks : A new deep architecture. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI 2011)*, pages 337–346, 2011.

I. M. Hugo Larochelle. The neural autoregressive distribution estimator. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, volume 15, pages 29–37, Ft. Lauderdale, USA, 2011. JMLR W&CP.

M. M. B. X. D. W.-F. S. O. A. C. Y. B. Ian Goodfellow, Jean Pouget-Abadie. Generative adversarial nets. pages 2672–2680, 2014.

F. B. P. L. R. P. G. D. J. T. D. W.-F. Y. B. James Bergstra, Olivier Breuleux. Theano : a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

Y. S. John Duchi, Elad Hazan. Adaptive subgradient methods for online learning and stochastic optimization. Rapport technique, EECS Department, University of California, Berkeley, Mar 2010.

A. M. C. B. D. W. Karol Gregor, Ivo Danihelka. Deep AutoRegressive Networks. In *Proceedings of the 31th Annual International Conference on Machine Learning (ICML 2014)*, pages 1242–1250. JMLR.org, 2014.

Y. L. Karol Gregor. Learning representations by maximizing compression, 2011. arXiv :1108.1169v1.

Y. B. Laurent Dinh, David Krueger. NICE : non-linear independent components estimation. *CoRR*, abs/1410.8516, 2014.

S. Z. Y. L. R. F. Li Wan, Matthew D. Zeiler. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning,(ICML 2013)*, pages 1058–1066, 2013.

BIBLIOGRAPHIE

A. K. I. S. R. S. Nitish Srivastava, Geoffrey Hinton. Dropout : A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15 :1929–1958, 2014.

K. H. P. Baldi. Neural networks and principal component analysis : Learning from examples without local minima. *Neural Netw.*, 2(1) :53–58, January 1989. ISSN 0893-6080.

F. Rosenblatt. The perceptron–a perceiving and recognizing automaton. Rapport technique, Cornell Aeronautical Laboratory, 1957.

R. Salakhutdinov and I. Murray. On the quantitative analysis of deep belief networks. In *Proceedings of the 25th Annual International Conference on Machine Learning (ICML 2008)*, pages 872–879. Omnipress, 2008.

P. Smolensky. Parallel distributed processing : Explorations in the microstructure of cognition, vol. 1. pages 194–281, 1986.

R. S. Z. S. L. S. S. C. S. Tanya Schmah, Geoffrey E. Hinton. Generative versus discriminative training of RBMs for classification of fMRI images. In *Advances in Neural Information Processing Systems 21*, pages 1409–1416, 2009.

Ç. G. K. C. S. G. Y. B. Yann Dauphin, Razvan Pascanu. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572, 2014.

C. C. Yann LeCun. The mnist database of handwritten digits, 1998.

G. A. J. Y. Yoshua Bengio, Eric Laufer. Deep generative stochastic networks trainable by backprop. In *Proceedings of the 31th Annual International Conference on Machine Learning (ICML 2014)*, pages 226–234. JMLR.org, 2014.

S. B. Yoshua Bengio. Modeling high-dimensional discrete data with multi-layer neural networks. In *Advances in Neural Information Processing Systems 12 (NIPS'99)*, pages 400–406. MIT Press, 2000.

M. D. Zeiler. ADADELTA : an adaptive learning rate method, 2012. arXiv :1212.5701v1.